Sistemi Operativi T

Kevin Michael Frick

14 dicembre 2019

1 Organizzazione dei SO

1. Organizzazione dei SO: sistemi monolitici, modulari, microkernel.

Risposta: Un sistema operativo *monolitico* (e.g. il kernel Linux) è composto da un unico modulo che implementa tutte le procedure necessarie per eseguire i propri compiti. Un sistema simile è molto veloce nell'esecuzione dato il ridotto numero di chiamate a funzioni, ma diventa facilmente complesso e pesante.

Un sistema operativo *a livelli* (e.g. TED) è diviso in una serie di moduli progressivamente più astratti: il modulo a livello più basso implementa le chiamate di sistema più semplici le quali vengono utilizzate dalle chiamate implementate dal livello superiore e così via. Ogni livello realizza un insieme di funzionalità che vengono offerti, tramite interfacce ben definite, al livello superiore. Un sistema simile è più complesso da progettare e più lento dato il maggiore numero di chiamate di sistema e i vari livelli da attraversare: per questo motivo si cerca di limitare il più possibile il numero di livelli. Un sistema operativo modulare è facilmente estensibile aggiungendo moduli ulteriori o modificando quelli esistenti senza dover lavorare sui livelli di astrazione più bassi: ogni livello conosce quelli sottostanti solo tramite le interfacce fornite dal livello immediatamente inferiore.

Un sistema operativo a *microkernel* (e.g. Minix) implementa solo le chiamate di sistema strettamente necessarie per accedere alle risorse protette, mentre tutte le utilità di sistema ulteriori lavorano in modo utente e si servono delle funzioni del microkernel. Un sistema simile è molto affidabile, estendibile e semplice da progettare, ma è lento a causa del gran numero di chiamate di sistema necessarie per eseguire qualunque operazione, anche se molto semplice. I microkernel vengono spesso utilizzati per scopi didattici.

2 Allocazione della memoria centrale

2. Allocazione della memoria nei sistemi multiprogrammati.

Risposta: Nei sistemi multiprogrammati il sistema operativo svolge diversi compiti:

- **allocazione e deallocazione** della memoria ai processi;
- ***binding*** degli indirizzi logici a indirizzi fisici;
- **protezione** della memoria da accessi non autorizzati;

- realizzazione della **memoria virtuale**.

Gli indirizzi di memoria possono essere *simbolici* (e.g. nomi di variabili), *logici* o *fisici*. Un file contenente codice di programma che utilizza indirizzi simbolici viene convertito dal compilatore in un *file oggetto* che dopo la fase di *linking* viene collegato con le librerie di sistema utilizzate, convertendo gli indirizzi in indirizzi logici. Quando poi il sistema operativo carica oltre all'eseguibile rilocabile generato dal linker anche le librerie a caricamento dinamico (*dynamic link libraries*) gli indirizzi vengono tradotti in indirizzi fisici. La memoria virtuale è una funzione svolta dai sistemi operativi che permette di svincolare la capacità di esecuzione di programmi dall'effettiva disponibilità di memoria centrale: ogni processo dispone del proprio *spazio di indirizzamento virtuale* che può essere mappato nello spazio di indirizzamento fisico con una serie di politiche:

- **allocazione contigua a partizione singola**, lo spazio di indirizzamento virtuale è allocato in una porzione contigua dello spazio di indirizzamento fisico. Questa politica però vincola la dimensione della memoria virtuale alle dimensioni fisiche della memoria centrale e impedisce la multiprogrammazione;

- **allocazione contigua a partizione multipla**, a ogni processo viene allocata una diversa area di memoria nello spazio di indirizzamento fisico. Le partizioni possono avere dimensione fissa, nel qual caso però si ha *frammentazione interna* quando un processo non utilizza tutta la memoria allocatagli e si limitano le possibilità di multiprogrammazione, o a dimensione variabile, che permette di sfruttare un grado flessibile di multiprogrammazione ed elimina la frammentazione interna creando però *frammentazione esterna* man mano che i processi vengono avviati e terminati, necessitando quindi di periodica *compattazione*;

- **allocazione non contigua**, nella quale gli spazi di indirizzamento virtuali sono distribuiti in sezioni non contigue di memoria centrale e secondaria. L'allocazione non contigua può essere realizzata tramite *paginazione*, *segmentazione* o una combinazione delle due:

  - la **paginazione** prevede il mapping di porzioni di memoria virtuale in posizioni non contigue della memoria fisica: ogni porzione di memoria virtuale, di dimensione costante, è detta *pagina* o *frame*. Le pagine possono essere memorizzate anche in memoria secondaria, nel qual caso vengono caricate in memoria centrale, su richiesta, dal *pager*. Un pager che non carica una pagina in memoria centrale prima che essa venga richiesta è detto "pigro" (*lazy pager*). È possibile che non vi sia abbastanza spazio in memoria secondaria per caricare una pagina richiesta: in tal caso si parla di *page fault* e una *pagina vittima* viene eliminata (ma lo swap-out viene eseguito solo se è stata modificata, in modo da risparmiare operazioni sulla memoria secondaria) per fare posto a quella richiesta. Il sistema operativo mantiene una struttura dati detta *tabella delle pagine* nella quale a ogni indirizzo di pagina è associato un indirizzo di memoria fisica. All'interno di ogni pagina gli indirizzi sono contigui. La tabella delle pagine può diventare molto grande al crescere dei processi, perciò si utilizza la *paginazione a più livelli*: anche la tabella delle pagine è paginata, e le sue pagine possono essere paginate a loro volta. Il tempo di accesso alla memoria virtuale cresce con i livelli di paginazione. Esiste anche una struttura dati particolare, detta *tabella delle pagine invertita*, che memorizza in una tabella gli indirizzi delle pagine e il *pid* del processo che le sta usando. La ricerca in questa struttura però è lenta e l'uso della tabella di pagine

invertita rende difficile la condivisione di memoria virtuale tra processi. Può capitare che un processo impieghi più tempo a paginare che ad eseguire (fenomeno di *thrashing*): per far fronte a ciò ci si serve di pager che precaricano in memoria centrale l'insieme di pagine che si prevede verranno usate da un processo durante la sua esecuzione (*working set*);

- la **segmentazione** prevede la partizione dello spazio di indirizzamento virtuale in segmenti di dimensione variabile, identificati da un nome e divisi semanticamente (e.g. *codice*, *dati*, *heap*, *stack*). All'interno di ogni segmento i dati sono allocati in modo contiguo e non è fissato un ordine tra i segmenti: ognuno è identificato da un intero a cui il SO fa riferimento.

Spesso si utilizza una combinazione di queste due tecniche: lo spazio di indirizzamento virtuale è segmentato e ogni segmento è paginato.

3. Significato e trattamento dei *page fault*.

Risposta: Il *paging* è una tecnica che permette di gestire la memoria virtuale di un processo allocandola in posizioni non contigue della memoria fisica: ogni porzione di memoria virtuale, di dimensione costante, è detta *pagina* o *frame*. Si ha un *page fault* quando una pagina richiesta da un processo non risiede in memoria centrale ma in memoria secondaria. Se c'è spazio sufficiente in memoria centrale la pagina richiesta viene allocata, altrimenti viene rimossa una *pagina vittima*. La pagina vittima può essere scelta secondo una serie di politiche:

- **Least Recently Used**, viene deallocata la pagina usata meno recentemente;
- **FIFO**, viene deallocata la pagina allocata per prima;
- **Least Frequently Used**, viene deallocata la pagina usata meno frequentemente.

Per evitare i *page fault* alcuni sistemi operativi si servono di politiche di *preallocazione*, allocando preventivamente alcune pagine secondo il *principio di località spaziale* (un processo tende ad accedere ad aree di memoria vicine tra loro) e *temporale* (un processo accederà con probabilità maggiore alle aree di memoria vicine tra loro). Lo spazio di memoria preallocato secondo il principio di località viene detto *working set*.

3 Allocazione della memoria secondaria

4. Organizzazione logica del file system Unix.

Risposta:

Il file system Unix gestisce tre tipi di file:

- **file regolari**;
- **direttori**;
- **dispositivi a blocchi**, file speciali residenti nel direttorio `/dev`.

Anche i direttori, quindi, sono rappresentati da file in Unix. Il file system Unix è strutturato come un grafo aciclico: ogni direttorio può contenere sottodirettori e possono esserci più collegamenti allo stesso file.

A ogni file sono associati dodici bit di protezione: i primi nove permettono di assegnare o revocare i diritti di lettura, scrittura ed esecuzione rispettivamente al proprietario (User), al gruppo di cui fa parte (Group) e agli altri utenti (Others). Gli altri tre bit permettono di attivare o disattivare le proprietà *suid*, *sgid*, *sticky*: *suid* permette a chi esegue il file di "impersonare" l'utente proprietario del file; *sgid* permette a chi esegue il file di "impersonare" un membro del gruppo del proprietario del file; *sticky* permette al programma di rimanere in area di swap dopo che ha terminato la propria esecuzione in modo da poter essere riavviato più velocemente.

5. Organizzazione fisica del file system Unix.

Risposta: Un dispositivo formattato in Unix è suddiviso in quattro parti:

- **boot block**, contenente le informazioni necessarie per l'avvio del SO;
- **super block**, contenente informazioni sul numero di blocchi e la loro dimensione;
- **data blocks**, l'area in cui vengono effettivamente allocati i blocchi di dati;
- **ilist**, contenente la lista di file e direttori (detti *inode*) indicizzati tramite gli *inumber*.

Ogni file è identificato da un *inode*, che contiene informazioni quali il nome, il proprietario, la data di ultima modifica oltre ai 12 bit di protezione. I file non sono allocati in maniera contigua: l'*inode* di un file contiene informazioni sui blocchi sui quali è memorizzato nel *vettore di indirizzamento*, costituito da una serie di indirizzi che consentono di individuare i blocchi in cui è allocato il file. L'indirizzamento di Unix è *a più livelli*: gli ultimi puntatori del vettore di indirizzamento puntano a blocchi che contengono a loro volta altri indirizzi. Un blocco fisico su un disco è indicizzato da tre parametri:

- **F**, il numero della faccia sul disco;
- **T**, il numero della traccia all'interno della faccia;
- **S**, la posizione del settore all'interno della traccia.

L'insieme dei settori che compongono un disco può essere trattato come un array lineare di blocchi, nel quale l'indirizzo `i` di un blocco alla posizione (f, t, s) è pari a $i = M \cdot N \cdot f + N \cdot t + s$ indicando con M il numero di tracce per faccia e con N il numero di settori per traccia.

6. Politiche di *scheduling* dei dischi.

Risposta: Le richieste di accesso ai settori possono essere soddisfatte secondo una serie di politiche:

- **First Come First Served**, le richieste vengono gestite nell'ordine in cui arrivano;

- **Shortest Seek Time First**, viene stimato il tempo richiesto per portare la testina a un dato disco (*seek time*) e le richieste con *seek time* minore vengono gestite per prime;
- **SCAN**, la testina esplora il disco e soddisfa le richieste non appena incontra i settori richiesti;
- **CSCAN**, una variante della politica SCAN volta a ridurre i tempi di gestione delle richieste scansionando sempre nella stessa direzione, in modo da ridurre i tempi di attesa dato che le richieste più "vecchie" si troveranno al capo opposto del disco rispetto a quello in cui termina l'algoritmo SCAN.

4 Processi e thread

7. Scheduling della CPU in sistemi interattivi.

Risposta: Nei SO convivono tre livelli di *scheduling*: quello a lungo termine, nei sistemi batch, si occupa di decidere quali processi vengono eseguiti; quello a medio termine si occupa di trasferire i processi dalla memoria centrale a quella secondaria quando non stanno eseguendo (*swap-out*) e viceversa (*swap-in*). Lo scheduler a breve termine, invece, si occupa di allocare la CPU ai vari processi già in esecuzione e in memoria centrale. La politica di scheduling influenza direttamente il numero di cambi di contesto e, quindi, l'*overhead* del SO. Vi sono una serie di politiche diverse per lo scheduler a breve termine. Alcune sono più adatte per i sistemi batch, mentre per i sistemi interattivi è necessario uno scheduler in grado di minimizzare il tempo di *attesa* (ovvero il tempo atteso dai programmi per avere la CPU allocata) e il tempo di *risposta*, ovvero il tempo che intercorre tra un comando dell'utente e la risposta del programma.

Gli scheduler a breve termine possono avere possibilità di prelazione (*preemptive*) o meno (*non-preemptive*). Uno scheduler con prelazione può revocare la CPU ad un processo una volta assegnatagli, cosa non possibile in un sistema non-preemptive. Le richieste continuamente variabili dei sistemi interattivi richiedono spesso scheduler con prelazione. Alcune delle politiche di *scheduling* più comuni sono:

- **First Come First Served**, i *job* usano la CPU nell'ordine in cui la richiedono. Questo è un algoritmo senza prelazione non utilizzabile nei sistemi interattivi;
- **Shortest Job First**, viene stimata una durata dei *job* e viene data la priorità a quello che ha il tempo di esecuzione minore. Questo *scheduler* senza prelazione è ottimale, ovvero minimizza il tempo necessario per l'esecuzione dei processi, ma non potendo disporre di una durata esatta per un processo è necessario stimarla;
- **Shortest Remaining Time First**, viene periodicamente stimata una durata dei *job* e viene data la priorità a quello che ha il tempo di esecuzione minore. Se un processo che entra in coda ha una durata minore di quella residua stimata del processo attualmente in esecuzione la CPU viene assegnata al nuovo processo. Questo scheduler può essere visto come una variante con prelazione del SJF;
- **Round Robin**, ogni *job* può utilizzare la CPU per un tempo predeterminato, terminato il quale essa viene assegnata ad un altro *job*. Questo è un algoritmo con prelazione molto utilizzato nei sistemi interattivi che tuttavia introduce un notevole overhead a causa dei frequenti cambi di contesto;

- **con priorità**, a ogni *job* si assegna una priorità e i processi con priorità più alta possono utilizzare la CPU più frequentemente e/o per un tempo maggiore. Ciò può creare problemi di *starvation* dei processi con priorità bassa che non riescono mai a completare la propria esecuzione; per far fronte a ciò la priorità dei processi viene aumentata man mano che rimangono in coda (*aging*). Uno scheduler con priorità può essere definito internamente (ovvero assegnando la priorità in base a caratteristiche intrinseche di ogni processo, come nel caso dello SRTF) o esternamente (assegnando la priorità ai processi secondo fattori esterni, e.g. lasciandola definire all'utente) e statico (la priorità dei processi è fissa) o dinamico.

Spesso vengono combinati più scheduler, ad esempio i sistemi interattivi solitamente utilizzano uno scheduler detto **Multiple Level Feedback Queues**: vi sono più code nelle quali risiedono i processi di diverso tipo (foreground, background, batch ecc.), ognuna con una diversa priorità e un diverso algoritmo di scheduling e i processi possono muoversi da una coda all'altra secondo la propria storia (es. un processo che ha utilizzato da molto tempo la CPU vede la sua priorità calare).

8. Interazione tra processi nel modello ad ambiente globale (detto anche a memoria comune).

Risposta: Nel *modello a memoria comune* i processi condividono uno spazio di indirizzamento. Ogni applicazione è rappresentata da un insieme di *processi* (componenti attive) e *risorse* (componenti passive). I processi comunicano tramite l'accesso a risorse comuni. In questo modello, i processi e i thread vengono trattati allo stesso modo dato che entrambi condividono il proprio spazio di indirizzamento con altri processi/thread.

Le risorse comuni devono gestire l'accesso in maniera *mutualmente esclusiva*: mentre un processo opera su una risorsa comune (in una *sezione critica* del proprio codice) la risorsa non deve essere accessibili da altri processi (cfr. domanda 11). È inoltre necessario evitare *deadlock*, situazioni di stallo nelle quali nessun processo può proseguire la propria esecuzione: ogni processo è in attesa di un evento che può eseguire solo un altro processo. Si verifica un deadlock **se e solo se** si ha:

1. **mutua esclusione**: un solo processo alla volta può accedere alle risorse comuni;
2. **possesso e attesa**: un processo può richiedere un'altra risorsa mentre ne sta occupando una;
3. **assenza di prelazione**: una risorsa non può essere tolta a un processo che la sta occupando;
4. **attesa circolare**: dato l'insieme di processi $\{P_1, ..., P_n\}$ ogni processo $P_k, k < n$ necessita di una risorsa da P_{k+1} e P_n necessita una risorsa da P_1.

Se anche una sola delle condizioni non è verificata non si verifica deadlock. Le soluzioni al problema della mutua esclusione (cfr. domanda 11) devono tenerne conto ed evitare possibilità di deadlock.

9. Interazione tra processi nel modello ad ambiente locale.

Risposta: Nel *modello ad ambiente locale*, secondo il quale gli spazi di indirizzamento dei processi sono rigidamente separati, i processi interagiscono esclusivamente *comunicando* tramite meccanismi di IPC (*Inter-Process Communication*). La comunicazione tra processi avviene tramite due primitive, `send` e `receive`. La comunicazione può essere diretta o indirietta. Nel caso della comunicazione diretta si può avere un meccanismo *simmetrico*, nel quale il canale di comunicazione utilizzato è univoco per ogni coppia di processi; se invece il meccanismo è *asimmetrico* un processo può ricevere messaggi da più processi e la `receive` prende in ingresso anche il *pid* del processo dal quale intende ricevere un messaggio. Nel caso di comunicazione indiretta il sistema operativo gestisce delle *porte* (o *mailbox*) e ogni `send` o `receive` prende in ingresso, oltre al messaggio, anche la *mailbox* a cui inviarlo o da cui prelevarlo.

Per quanto concerne la sincronizzazione, la `send` può essere *sincrona*, *asincrona* o *a chiamata di procedura remota* mentre la `receive` può essere o non essere *bloccante*. Una `send` sincrona blocca il processo che invia il messaggio finché il ricevente non chiama la `receive`; una `send` asincrona permette invece al processo di proseguire la propria esecuzione; una `send` a chiamata di procedura remota, invece, richiede un servizio dal processo destinatario e permette la prosecuzione dell'esecuzione solamente dopo che il destinatario ha chiamato la procedura e ne ha comunicato l'esito. Una `receive` bloccante blocca il processo che tenta di riceve un messaggio nel caso in cui non ci siano messaggi da ricevere, mentre una non bloccante permette al processo di proseguire la propria esecuzione qualora non ci siano messaggi.

10. Rappresentazione dei processi Unix.

Risposta: Nel sistema operativo Unix i processi seguono il *modello a memoria locale* e interagiscono solo tramite comunicazione. Ogni processo ha il proprio spazio di indirizzamento, ma i processi Unix sono *rientranti* ovvero possono condividere codice. Le informazioni normalmente contenute nel descrittore di un processo (PCB, *Process Control Block*) sono suddivise in due strutture: la *Process Structure* e la *User Structure*. La PS deve rimanere in memoria centrale per tutta la durata del processo mentre la US è swappable.

La PS contiene le informazioni necessarie per la gestione del processo anche quando esso è *swapped*: il *pid* del processo, riferimenti alle sue aree dati/stack, informazioni sul suo stato (init, ready, running, waiting, terminated, zombie o swapped - *zombie* e *swapped* sono due stati particolari che Unix aggiunge al modello a cinque stati e che indicano, rispettivamente, un processo che è terminato e attende che il padre si accorga della sua terminazione e un processo spostato dalla memoria centrale a quella secondaria dallo scheduler di medio termine), la sua priorità, il puntatore alla US e il puntatore al prossimo processo nella coda corrispondente al proprio stato. La US contiene le informazioni che non sono necessarie quando il processo è *swapped* e che possono essere quindi allocate in memoria secondaria a loro volta: una copia dei registri CPU, informazioni sulle risorse allocate e sui segnali, il direttorio corrente, informazioni sul proprietario e sul suo gruppo.

Le corrispondenze tra ogni processo e la sua PS/US sono conservate nella *tabella dei processi*, che contiene una voce per ogni processo aperto. Il kernel gestisce inoltre una tabella del codice, detta *text table*, che contiene il codice eseguito dai vari processi e i pid dei processi che lo stanno utilizzando. Nel PCB, quindi, il codice è espresso mediante un riferimento a una voce della text table. L'immagine di un processo è dunque composta da PCB, aree dati e stack e da una voce della text table (detta *text structure*). Il kernel mantiene inoltre una *text structure* contenente il codice delle chiamate di sistema.

11. Possibili soluzioni al problema della mutua esclusione.

Risposta: Le soluzioni al problema della mutua esclusione possono essere di tipo *algoritmico*, *hardware* o *software*. Le soluzioni al problema della mutua esclusione hanno tre requisiti da rispettare:

1. **mutua esclusione**;
2. un processo all'esterno di una sezione critica non può rendere **impossibile ad altri processi** eseguire sezioni critiche;
3. assenza di **deadlock**.

Due possibili soluzioni algoritmiche sono l'*algoritmo di Dekker* e l'*algoritmo di Peterson*; entrambe si basano su tre variabili `busy1`, `busy2` e `turn`. Seguendo l'algoritmo di Dekker, il primo processo ad arrivare alla propria sezione critica assegna un valore vero a `busy1` (segnalando *volontà di eseguire la sezione critica*) ed entra in un ciclo `while` di *attesa attiva*: se `busy2` è vera, imposta `busy1` a 0, attende (con un secondo ciclo `while` di attesa attiva) la fine del turno dell'altro processo e in seguito riassegna un valore vero a `busy1`. Dato che questi controlli si trovano in un ciclo `while`, prima di entrare nella propria sezione critica il processo ricontrolla che l'altro non stia eseguendo la propria. Se è libero di proseguire, il processo esegue la propria sezione critica e dopo averla eseguita assegna il turno all'altro processo e imposta `busy1` a 0.

L'algoritmo di Peterson è simile, ma vi è un ciclo di attesa attiva in meno. Inoltre è possibile utilizzarlo per sincronizzare n processi (l'algoritmo di Dekker funziona solo con due) e risolve il problema della *starvation* ovvero assicura che ogni processo possa eseguire la propria sezione critica entro un tempo finito. Ogni processo, secondo questo algoritmo, prima di entrare nella sezione critica imposta la propria variabile `busyX` a 1, imposta `turn` in modo che sia il turno dell'*altro* processo, poi entra in un ciclo di attesa attiva che termina quando l'altro processo ha terminato la propria sezione critica (`busyY = 0`) e assegnato il turno al primo processo. Il processo esegue quindi la propria sezione critica e imposta `busyX` a 0. Ciò assicura che un processo possa eseguire se e solo se altri non stanno *già eseguendo* la propria sezione critica.

```
busy1 = true;
while (busy2) {
        busy1 = 0;
        while (turn != 1) {
        // attesa attiva
        }
        busy1 = true;
}
// sezione critica
turn = 2;
busy1 = 0;
```

Algoritmo di Dekker.

```
busyX = true;
turn = Y;
while (turn == y && busyY) {
// attesa attiva
}
// sezione critica
busyX = 0;
```

Algoritmo di Peterson.

Una possibile soluzione hardware è disabilitare le interruzioni mentre un processo sta eseguendo la propria sezione critica. Ciò assicura che la CPU non interrompa mai un processo all'interno di una sezione critica, ma rende anche l'intero sistema insensibile agli eventi esterni durante qualunque sezione critica, rende impossibile la concorrenza e non è utilizzabile nei sistemi con più CPU: mentre un processo che esegue sulla CPU X disabilità le interruzioni mentre accede ad una risorsa, le interruzioni rimangono abilitate sulla CPU Y che può eseguire un processo che richiede la stessa risorsa.

Una seconda soluzione hardware si basa sull'esistenza, nell'ISA del processore, dell'istruzione atomica `tsl`, *test-and-set*, la quale permette di controllare il valore di verità di una variabile e cambiarlo nello stesso ciclo di memoria. Questa istruzione permette di implementare una struttura dati detta *lock*, che viene *chiusa* prima di ogni sezione critica e *aperta* alla fine. Prima di entrare nella sezione critica, ogni processo controlla (tramite test-and-set) che il lock sia aperto e se lo è lo chiude nello stesso ciclo di memoria per poi proseguire nella propria sezione critica. La test-and-set impedisce corse critiche del tipo:

1. P1 controlla il lock e lo trova aperto
2. P2 controlla il lock e lo trova aperto
3. P1 chiude il lock
4. P2 chiude il lock

che permetterebbero sia a P1 che a P2 di eseguire la propria sezione critica nello stesso momento.

Soluzioni software possibili sono il *semaforo* e il *monitor* (cfr. domande 14 e 15).

12. Comunicazione tra processi Unix.

Risposta: I processi in Unix comunicano con *socket* (fra processi nella stessa rete), *pipe* (fra processi nella stessa gerarchia) e *fifo* (tra processi sulla stessa macchina).

Una *pipe* è un canale bidirezionale di comunicazione aperta tramite la *system call* `pipe`. Una pipe è un canale unidirezionale (accessibile a un estremo in lettura e a un altro in scrittura) molti-a-molti (più processi possono condividere la stessa pipe). Ogni processo figlio del creatore della pipe ha accesso a due *file descriptor* che permettono di scrivere o leggere sulla pipe. Non c'è naming esplicito del destinatario (modello a *mailbox*) e la comunicazione è asincrona: una pipe immagazzina messaggi (fino a una capienza massima) finché essi non vengono letti da un altro processo. Un processo può leggere da o scrivere sulla pipe come da un qualunque altro file descriptor e chiudere l'accesso in lettura o in scrittura con le stesse chiamate di sistema. Una pipe realizza automaticamente la sincronizzazione: un processo che scrive su una pipe piena o legge da una vuota si sospende automaticamente.

Una *fifo* è una pipe realizzata con un file che permette la comunicazione anche tra processi non appartenenti alla stessa gerarchia. La comunicazione è unidirezionale e *first-in-first-out*. Una *fifo* viene creata con la *system call* `mkfifo` che prende in ingresso il nome del file dove sarà memorizzata. Lettura e scrittura sulla fifo avvengono come un qualunque altro file descriptor.

13. I thread: caratteristiche e realizzazione.

Risposta: I thread, o processi leggeri, sono processi che rappresentano un singolo *flusso di esecuzione* all'interno di un processo normale (detto *pesante*) e condividono tra di loro e con il processo pesante codice, dati e spazio di indirizzamento. Non possedendo risorse è molto facile creare e distruggere i thread. Nei SO tradizionali (e.g. UNIX) ogni processo ha un solo thread; altri SO, detti *multithreaded*, sono in grado di associare più thread a ogni processo.

La gestione dei thread può avvenire a livello kernel o a livello utente. Nel caso di gestione a livello utente si ha una libreria di funzioni che si occupa di creare, eseguire e distruggere i thread. Ogni processo parte con un solo thread e può, senza richiami a chiamate di sistema, crearne altri. Il SO ignora completamente i thread, vedendo solo processi, e quando il processo principale viene sospeso anche tutti i suoi thread vengono sospesi. Questo metodo di gestione può essere implementato anche in sistemi che non supportano nativamente i thread (e.g. UNIX) ed è più efficiente dato che l'intera gestione dei thread non richiede chiamate di sistema, ma non è in grado di sfruttare il parallelismo offerto dai sistemi multiprocessore. In caso di gestione dei thread a livello kernel, invece, a ogni funzione di libreria corrisponde una system call e il sistema è in grado di distinguere thread e processi, applicando il proprio scheduler a entrambi. Questo sistema è meno efficiente dato il maggiore numero di chiamate di sistema ma permette di sfruttare il parallelismo in caso di sistemi multiprocessore, eseguendo thread diversi di uno stesso processo in parallelo.

In Java essi sono realizzati tramite la classe `Thread` e l'interfaccia `Runnable`. Una classe può estendere la prima o implementare la seconda: in entrambi i casi va implementato il metodo `run` che implementa le azioni del thread. Per avviare una classe estensione di `Thread` è sufficiente chiamare il metodo `start`, mentre per avviare una classe che implementi `Runnable` è necessario creare un nuovo `Thread` tramite l'operatore `new` al cui metodo `start` passare la classe come argomento.

14. Il semaforo.

Risposta: Un semaforo è una particolare struttura dati che permette di implementare semplicemente meccanismi *produttore-consumatore* e di mutua esclusione. Esso consiste in una variabile intera non-negativa `value`, e due procedure `up` e `down`. Quando un processo vuole operare sulla risorsa gestita dal semaforo esso chiama la funzione `down`. Se `value` è non nullo, il processo lo decrementa, esegue le proprie operazioni e alla fine chiama la funzione `up`. In caso contrario il processo si mette in attesa finché non viene chiamata la funzione `up` da un altro processo che ha al momento accesso alla risorsa. `up` incrementa `value`.

Le due operazioni sono *atomiche* ovvero vengono eseguite in un'unica soluzione indivisibile. Questa proprietà permette di evitare corse critiche che possono sorgere se, ad esempio, due processi chiamano `down` contemporaneamente.

Un semaforo è realizzato tramite un intero non negativo che può avere un valore iniziale non nullo e una coda FIFO contenente i processi in attesa.

15. Il monitor.

Risposta: Il monitor è un costrutto realizzato dal compilatore che permette di gestire in maniera mutualmente esclusiva l'accesso ad una risorsa da parte di più processi. Oltre a risolvere il problema della mutua esclusione un monitor permette anche di assegnare diverse priorità ai processi che vogliono accedere alla risorsa mediante l'uso di *variabili condizione*.

Un monitor è realizzato con una classe che implementi alcuni metodi, detti *entry*, che permettono l'accesso alla risorsa in maniera mutualmente esclusiva: quando un processo sta chiamando un metodo entry, nessun altro processo può accedere alla risorsa. Se un processo tenta di accedere a un monitor mentre un altro sta eseguendo un metodo entry, esso si sospende nella *entry queue*, che gestisce le richieste di entrata in maniera FIFO.

Le variabili condizione sono variabili che vengono controllate prima di permettere l'accesso alla risorsa: se al momento in cui un processo tenta di accedervi la risorsa è occupata, esso si mette in attesa nella coda corrispondente alla propria priorità (*condition queue*) *e libera il monitor*. Un processo può svegliare gli altri processi in attesa in una coda tramite il metodo `signal`. Vi è però un problema: un processo potrebbe chiamare la `signal` e poi continuare ad accedere alla risorsa. Vi sono due possibili soluzioni: la semantica *signal-and-continue*, adottata da Java, prevede che un processo che chiami la `signal` svegli il primo processo segnalato che passa dalla condition queue alla entry queue; la semantica *signal-and-wait* invece prevede che non appena un processo chiami la `signal` esso si sospenda nella entry queue facendo accedere alla risorsa il processo segnalato. Come variante della *signal-and-wait* si ha la *signal-and-urgent-wait*, che prevede che il processo segnalante non si sospenda nella entry queue ma in un'altra coda, detta *urgent queue*, che ha priorità sulla entry queue. Quando una *signal-and-urgent-wait* corrisponde ad una istruzione di ritorno dal metodo entry il processo segnalato entra nel monitor *senza rilasciare la mutua esclusione*. È possibile inoltre risvegliare tutti i processi in attesa mediante la chiamata `signal_all` che risveglia tutti i processi in attesa in una data condition queue e li sposta nella entry queue.

5 Input/Output

16. Gestione dell'I/O: interazione tra processo e periferica nel caso di dispositivo abilitato alle interruzioni.

Risposta: Per gestire i sistemi abilitati alle interruzioni il SO assegna a ognuno di essi un semaforo inizializzato a 0. Quando un processo deve leggere un dato dal dispositivo esegue una `down` su questo semaforo, sospendendosi quindi se il dato non è al momento disponibile. Quando il dispositivo manda in segnale di *interrupt* alla CPU l'*handler* preposto si occupa di chiamare la primitiva `up` del semaforo. Il processo può quindi proseguire con il prelievo del dato dal dispositivo.

Quando al dispositivo non è richiesto un dato solo ma (ad esempio con un ciclo `for`) una serie di n dati è inefficiente risvegliare il processo che interagisce con il dispositivo per poi sospenderlo subito nuovamente. Si definisce quindi una struttura dati apposita, detta *descrittore del dispositivo*, che assieme alle funzioni associate del processo applicativo e di gestione delle interruzioni costituisce il *driver del dispositivo*. La funzione di lettura dal dispositivo ogniqualvolta viene richiesta una serie di dati imposta una variabile di stato (detta *contatore*) all'interno del descrittore al numero di dati richiesti e inizializza un buffer per poi sospendersi. A quel punto la funzione di gestione delle interruzioni, quando viene chiamata, è in grado di discernere dal valore del contatore se l'interruzione è dovuta ad un errore, è in mezzo a un ciclo di lettura o ne è la conclusione. Nel primo caso viene svegliata la funzione di lettura e viene impostato un apposito codice di errore nel descrittore, che verrà poi restituito al processo chiamante; nel secondo caso l'handler memorizza nel buffer il dato letto e sposta la "testina" indicata da un apposito puntatore; al termine di un ciclo invece l'handler sveglia la funzione di lettura la quale restituisce il numero di dati letti.

Disclaimer: Questo documento può contenere errori e imprecisioni che potrebbero danneggiare sistemi informatici, terminare relazioni e rapporti di lavoro, liberare le vesciche dei gatti sulla moquette e causare un conflitto termonucleare globale. Procedere con cautela.